XiFinder and V0Finder documentation

```
StRoot/StSecondaryVertexMaker/ :
            StV0FinderMaker.cxx
            StV0FinderMaker.h
            StXiFinderMaker.cxx
            StXiFinderMaker.h

StRoot/StSecondaryVertexMaker/doc/ :
            docXiFinder.tex
```

— Julien Faivre
Strangeness group
julien.faivre@ires.in2p3.fr

Contents

List of Figures

1 What are the V0Finder and XiFinder ?

Let's first define the 3 different sorts of secondary vertices :

- Kink vertices : when a charged particle decays into a charged plus a neutral,
- V0 vertices : when a neutral particle decays into two charged particles,
- Xi vertices : when a charged particle decays into a charged plus a neutral, and then the neutral daughter decays itself into two charged particles. One can't say that a Xi vertex is a Kink followed by a V0, because both charged tracks (mother and daughter) have to be seen in the TPC in order to say that the vertex is a kink. This is not the case for Xi vertices, because the $c\tau$ of particles who do a Xi vertex if much shorter than the distance between the primary vertex and the TPC (50 cm), even shorter than the distance to the first layer of the SVT (6.7 cm, versus $c\tau_{\Xi} = 4.9$ cm and $c\tau_{\Omega} = 2.5$ cm).

People working on strange particles in the strangeness group need a piece of code that is able to reconstruct the primary strange particles from the tracks. Those strange particles can be divided into two groups :

- **V0's :** those particles (Λ, K^0_s) decay into two particles ($\Lambda \longrightarrow p\pi^-$, $K^0_s \longrightarrow \pi^+\pi^-$). We need to be able to search and find them using only the daughter's tracks (that's all we have !).
- **Xi's :** those particles (Ξ, Ω) decay into 1 charged particle – called bachelor – and a Λ ($\Xi \longrightarrow \Lambda\pi^-$, $\Omega \longrightarrow \Lambda K^-$). The Λ, as said in the previous item, decays into two particles. So here, we need to find all combinations of 3 charged particles that are possibly the daughters of a unique strange particle.

The algorithms of those codes are described in the section **??**.

From the beginning of STAR and until year 2002, the codes used were what I will call afterwards *exiam* and *ev0am*. They are Fortran codes, located in `pams/global/exi/exiam.F` and `pams/global/ev0/ev0_am2.F`, with other files that are necessary for the code to be run (basically subroutines and interfacing functions).

So here is the "old" way to do things : the BFC is run, and it calls the series of makers that it is supposed to call. Among those makers are `StXiMaker` and `StV0Maker` – the strangeness makers, with `StKinkMaker` – and they are run after nearly all the other makers of the BFC, since they need the tracks to be reconstructed and the primary vertex to be found. The V0Maker is run first, finds the V0's, and those feed the XiMaker, which tries to find Xi candidates for each V0. At a deeper level, inside the `Make()` function of both the XiMaker and the V0Maker, are called the Fortran PAMs, i.e. respectively `ev0am` and `exiam`. PAM means either "Plugable Analysis Module" or "Physics Analysis Module". I don't know if somebody knows which of the two it is !

The interfacing between the BFC (C++) and the PAMs (Fortran) won't be discussed here. If you want to know more, you can have a look at those files : `pams/global/exi/exiam.idl`, `pams/global/ev0/ev0_am2.idl`, `pams/idl/dst_track.idl`, `pams/idl/dst_vertex.idl`, `pams/idl/dst_v0_vertex.idl`, `pams/idl/dst_xi_vertex.idl`. The data are passed – from maker to maker, and also between a maker and the PAM it calls – by tables. For example the tracks are stored in a table, and ev0am will read the table to have the tracks parameters. It will then store the V0's in another table. Then, exiam will read this table and the table of tracks, and write the Xi's found in a third table. Classes for interfacing have a general name which is "St_tableName_Table.h", and they can be found in `include/tables/` (as examples : `St_exi_exipar_Table.h`, `St_ev0_aux_Table.h`, `St_dst_track_Table.h`, `St_dst_xi_vertex_Table.h`,

etc... Further information about St_dst_track_Table can be found at `root.cern.ch/root/html/TTable.html` in the section "Class description"). On the other hand, the corresponding structures are stored in `pams/-global/idl/`, in files like `exi_exipar.idl` (general name of the file : tableName.idl ; general name of the structure used afterwards : tableName_st). You could have a look at e.g. `StXiMaker::Init()` (the V0 and XiMaker's are in `StRoot/St_dst_Maker/`) to have an example of how all this is used.

As explained in the section **??**, the strangeness Fortran PAMs had to be replaced with C++ code. As exiam and ev0am refer to the corresponding PAMs, *XiFinder* and *V0Finder* are the names we gave to their C++ translations. They are makers, whose complete names are `StXiFinderMaker` and `StV0FinderMaker`. This is what we call the *strangeness StSecondaryVertexMaker package*.

2 Historical purpose for StSecondaryVertexMaker package

The StSecondaryVertexMaker package implements secondary vertex-finding in C++. *Why ?* There is and has been code for reconstucting secondary vertices in the form of the `ev0`, `exi`, and `tkf` PAMs. These PAMs were written in Fortran and work with tables. They are called from C++ makers currently kept in the St_dst_Maker package library. They work well, but suffer limitations :

- They cannot be re-run on DSTs after production,
- They cannot operate on the tracking output of ITTF.

In order to overcome both limitations, the preffered solution is to write C++ versions of these PAMs which can use StEvent structures for both input and output (versus trying to convert StEvent structures back into tables for input). This is the primary purpose of the StSecondaryVertexMaker package.

Advantages of being able to run such a code on the DSTs are that analysis such as rotating can be done without running the whole BFC on the daq files, as well as analysis that require modifications in the secondary vertex reconstruction code, like the value of the reconstruction cuts for example. The time profit is huge, since it takes more than 20 times more time to run the whole reconstruction chain than just the C++ secondary vertex reconstruction makers.

3 Overall algorithms of the codes

Apart from some parts described below, the V0Finder and XiFinder are essentially the ev0 and exi PAMs rewritten from Fortran to C++, from a "tabelized" way of communicating to a standard object-oriented, mono-language code.

3.1 KinkFinder

To be written.

3.2 V0Finder

Cut parameters are initially requested from the database (time stamps determine what is the nature of the data, e.g. `p-p`, `Au-Au`, `d-Au`). Then, tracks which satisfy a set of cuts are chosen as candidates for

V0 daughters. The daughter candidates are then examined in pairs of negative and positive daughters to see if the tracks approach each other and pass a series of cuts to determine if they are consistent with a V0 secondary decay.

Formerly, a second pass was required on the V0s. This was to facilitate their use in finding Xi decays. V0s from Xi decays are *secondary* V0s, and thus do not originate from the primary vertex. This means looser cuts are necessary on these V0s than *primary* V0s. So, the first pass was made with the loose cuts, the Xi decays were found, and then the V0s were run through tighter cuts to remove unused ones which were inconsistent with being primary V0s. This second pass prevented the output of significant numbers of unnecessary V0 candidates.

This second pass has been replaced by a different mechanism. Now, for each V0 which passes the looser secondary V0 cuts, a `UseV0()` function is called. The idea is that a XiFinder can be written which inherits from the V0Finder, and implements the `UseV0()` function to find Xi candidates with a given V0. The `UseV0()` function then returns true or false depending on whether any Xi candidates are found using that V0. Upon returning to the V0Finder code, the V0 is discarded if it neither passes the cuts for a primary V0 nor gets used in the `UseV0()` function.

This scheme has the advantage of not inserting a V0 into the StEvent vector of V0s unless it is a viable candidate. It also reduces considerably the memory overhead required during V0-finding as not all of the secondary V0 candidates are found and stored at once (particularly poignant in high-multiplicity events where many thousands of secondary V0s are considered). This only disadvantage is some overhead in making a function call in the middle of the V0-finding loop.

FIG. **??** shows that even if the code of the V0Finder is quite long, the alorithm is definitely simple.

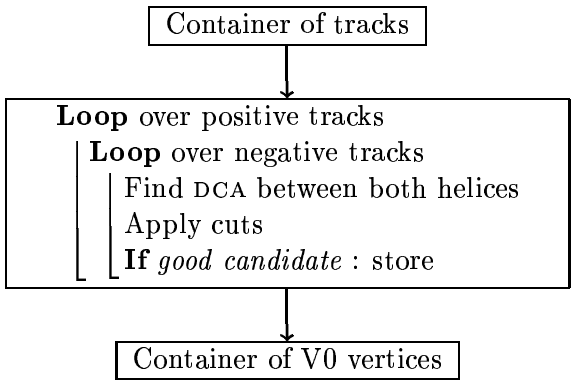


FIG. 1 : *Overall algorithm of the V0Finder.*

3.3 XiFinder

The `StXiFinderMaker` inherits from `StV0FinderMaker` as indicated above. Because it is actually a V0Finder itself via this inheritance, one need not instantiate a `StV0FinderMaker` if one instantiates a `StXiFinderMaker`.

Similar to the `StV0FinderMaker`, appropriate cut parameters are initially requested from the database. The same tracks that are considered for the V0s are also used as daughter candidates for the Xis. The `Make()` member function then simply calls the inherited `Make()` member function from the `StV0FinderMaker`. Control comes back to the `StXiFinderMaker` at the call to `UseV0()`, which is implemented here as a loop over Xi daughter candidates to be paired with the V0 daughter candidate. If the daughters approach each other and pass a series of cuts, the candidate is accepted and stored in

StEvent. Control is then passed back to the V0Finder, with a return value indicating whether the V0 was in fact used.

As for the V0Finder, FIG. **??** below shows that the XiFinder alorithm is very simple, although the code is long. More detailed algorithms can be found in section **??**.

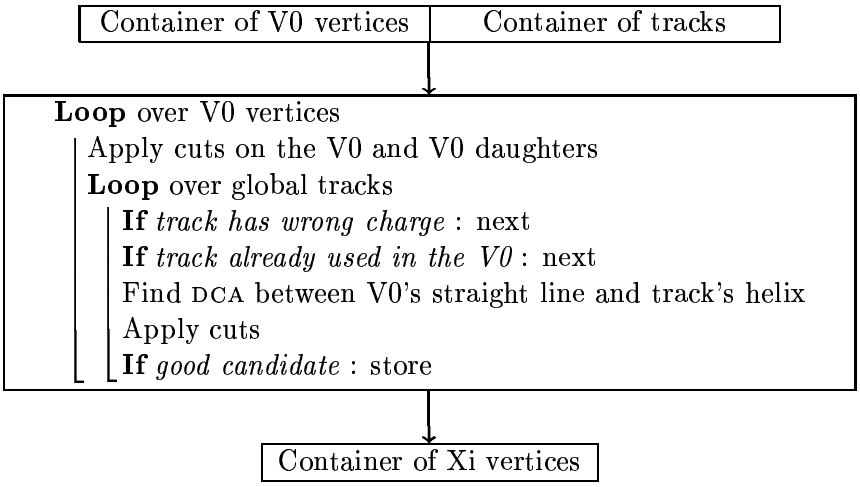


FIG. 2 : *Overall algorithm of the XiFinder.*

4 Structure of the Fortran and C++ codes

4.1 KinkFinder

To be written.

4.2 Common remarks for the V0/XiFinder

The first thing to mention is a change in the interaction between the V0Finder and the XiFinder. The XiFinder actually uses the V0's that have first been found by the V0Finder. The table below shows how this is done in the Fortran code :

Call `StV0Maker::Make()`	Finds V0's and store them in a table
Call `StXiMaker::Make()`	Loop over the V0's in the table to find Xi candidates
Call `StV0Maker::Trim()`	Loop over the V0's in the table to throw away the non-primary V0's that are not used in a Xi candidate

This waste of memory (storing V0's that will be deleted afterwards) and of time (scanning twice the table of V0's) is solved in the C++ code, by the fact that the class `StXiFinderMaker` actually inherits from `StV0FinderMaker`. The member-functions of each of them and their role are listed in FIG. **??**.

Note that `StXiFinderMaker::Init()` is the equivalent of `StV0FinderMaker::GetPars()`. The effect of the function `StV0FinderMaker::DontZapV0s` is that the V0's that are already in StEvent are kept, the V0's found by the V0Finder will be added. The function `StV0FinderMaker::UseExistingV0s`

also keeps the V0's that are already in StEvent, but also prevents the V0Finder to be run, and forces the XiFinder to use the V0's previously found. The function `StV0FinderMaker::UseITTFTracks` has been implemented for ITTF test purposes, since it allows the user to choose between using the ITTF tracks or using the tracks found by the standard reconstruction.

When the V0Finder is run alone, the method `StV0FinderMaker::UseV0()` is called for each V0 found, and returns `false`, since we don't want to find Xi's.

When both the XiFinder and V0Finder are run, the XiFinder first calls `StV0FinderMaker::Make()`. This function finds V0's and, for each of them, calls the `UseV0()` method. The latter runs the XiFinder algorithm, that will eventually tell `StV0FinderMaker::Make()` if the current V0 has been used in Xi candidates or not. This way to do, compared with the Fortran one, saves time and memory.

Function	Role in the V0Finder	Role in the XiFinder
`Init`	Inits	Gets `exipar` from the database
`Make`	V0Finder "central" algorithm	Calls the V0Finder
`Clear`	Clears	Not redefined
`GetPars`	Gets `ev0par2` from the database	Not redefined (not used)
`Prepare`	Finds event-wise parameters, fills tables	Not redefined
`UseV0`	Returns `false`	XiFinder "central" algorithm
`UseExistingV0s`	Sets a boolean flag	Not redefined
`DontZapV0s`	Sets a boolean flag	Not redefined
`UseITTFTracks`	Sets a boolean flag	Not redefined
`UsingITTFTracks`	Returns a boolean flag	Not redefined
`Trim`	Remove the V0's that don't pass cuts	Not redefined (not used)

FIG. 3 : *Member-functions of* `StV0FinderMaker` *and* `StXiFinderMaker`.

The table below shows how this is run in the C++ XiFinder (provided that it's the XiFinder that is called and not just the V0Finder) :

Call `StXiFinderMaker::Make()`	(Calls everything below)
Call `StXiFinderMaker::Prepare()`	Finds event-wise parameters and fills the tables
Call `StV0FinderMaker::Make()`	Finds V0's in this event
Call `StXiFinder::UseV0()`	Finds Xi's for a given V0 and store them in StEvent
(back in `V0FinderMaker::Make`)	If V0 is used in Xi's / may be primary : store in StEvent

4.3 V0Finder

FIG. **??** shows the detailed structure of the V0Finder, with all the cuts that are applied. Of course, more accurate information can be found... by looking at the code ;-) .

Before the loops, tables called `ptrks`, `ntrks`, etc..., are filled in the function `StV0FinderMaker::Prepare()`, and the values used afterwards are those that are stored in these tables, in order to improve the speed of the code. The XiFinder doesn't use all of them yet... but will.

The cuts' value are got in the function `StV0FinderMaker::GetPars()`, and stored in the member objects `pars` and `pars2`. Here are the components :

- `n_point` : number of hits,
- `dcapnmin` : distance of closest approach between the daughter tracks and the primary vertex,
- `dca` : distance of closest approach between the V0 daughters,

- `dlen` : decay length of the V0,
- `dcav0` : distance of closest approach between the V0 trajectory and the primary vertex,
- `alpha_max` : α Armanteros,
- `ptarm_max` : $p_\perp$ Armanteros.

Get parameters from database
Get event
Get position of the primary vertex
Loop over all tracks
  Select normal vs ITTF
  If *bad flag* : next
  If *bad detector* ID : next
  Store track and parameters in tables
    (separately pos. and neg.)
Loop over positive tracks
  Loop over negative tracks
    Determine V0's detector ID
    CUT on number of hits
    CUT on dcaTrackToPvx
    Find number of intersection points between both helices
    Find 2D dca between both helices at both intersection points
    Keep the smallest dca
    CUT if one track doesn't point away from Pvx
    CUT if V0 decays after first hit of either track
    Calulate approximated 3D dca between both helices
    CUT on dcaV0Daughters
    CUT on dcaTrackToPvx
    CUT on decay length from Pvx
    CUT if V0 doesn't point away from Pvx
    CUT on dcaV0ToPvx
    CUT on α_{Arm}
    CUT on $p_{\perp_{Arm}}$
    Fill an `StV0Vertex`
    Call `UseV0` to find if V0 is used for Xis
    If *primary or used in Xi* : store

FIG. 4 : *Cuts and algorithm of the V0Finder.*

Some information about each cut applied :

- Number of hits : both tracks must have a number of hits $\geq$ `DB`[0]`->n_point`
- DcaTrackToPvx : dca between each of the tracks and the primary vertex : see explanations in the next paragraph
- Track points away from Pvx : both tracks have to point away from the primary vertex, i.e. if we call X the primary vertex and M the point of a track that is the closest to the other helix, $\overrightarrow{p_M} \cdot \overrightarrow{XM}$ must be positive

[0]Database.

- V0 decays after the first hit of either track : this cut removes obviously bad candidates (or bad decay lengths calculations) which have a decay length that is e.g. longer than the size of the TPC. The first hit is assumed to be at `StPhysicalHelix::origin`. Calling it H, and V the V0 decay point, the requirement is that $\overrightarrow{p_{V0_{at\ V}}} \cdot \overrightarrow{VH}$ must be positive
- DcaV0Daughters : the calculated dca between both tracks has to be $<$ `DB->dca`
- DcaTrackToPvx : same as above, see also the next paragraph
- Decay length : the calculated V0 decay length has to be $>$ `DB->dlen`
- V0 points away from Pvx : calling X the primary vertex and A the point where both helices are closest to each other, $\overrightarrow{p_A} \cdot \overrightarrow{XA}$ must be positive
- DcaV0ToPvx : the calculated dca between the V0 and the primary vertex must be $<$ `DB->dcav0`
- Alpha Armanteros : α_{Arm} must be $\leq$ `DB->alpha_max`
- Pt Armanteros : $p_{\perp_{Arm}}$ must be $\leq$ `DB->ptarm_max`

Now, the dcaTrackToPvx cut requires some non-obvious explanations. Its second occurence is simple to understand : it is a cut on the dca of both tracks to the primary vertex – both have to be $>$ `DB->dcapnmin` – that is applied only when $p^2_{\perp_{V0}}$ is lower than a variable called `ptV0sq`, and whose value is set to $(3.5\ GeV)^2$ in the constructor of `StV0FinderMaker`. This is done to cut less signal at high-$p_\perp$, an area where there is very few background, thus enabeling a loosening of the cuts.

Its first occurence is done to apply this cut as soon as possible, for the code to be faster (less track pairs to process), at a time when $\overrightarrow{p_{V0}}$ is not calculated yet. The reason why this is possible is that, indexing with n (resp. p) what is related with the negative (resp. postive) daughter,

$$p_{\perp n} + p_{\perp p} \geq p_{\perp_{V0}} \tag{1}$$

Here is the demonstration : let's call r the axis that is parallel to $\overrightarrow{p_{V0}}$, θ the perpandicular axis. With these $(\overrightarrow{u_r}, \overrightarrow{u_\theta})$ coordinates, we have :

$$\left\{ \begin{array}{rcl} p_{V0_r} & = & p_{n_r} + p_{p_r} \\ p_{V0_\theta} & = & p_{n_\theta} + p_{p_\theta} = 0 \end{array} \right.$$

i.e. :

$$p_{\perp_{V0}} = p_{V0_r} = p_{n_r} + p_{p_r}$$

Since $p_{\perp n} \geq p_{n_r}$ and $p_{\perp p} \geq {p_{p_r}}$[0], we obtain :

$$p_{\perp n} + p_{\perp p} \geq p_{n_r} + p_{p_r} = p_{\perp_{V0}} \quad , \qquad \text{QED}^0$$

So the first occurence of the dcaTrackToPvx cut applies this cut when $(p_{\perp n} + p_{\perp p})^2 \leq$ `ptV0sq`, with `ptV0sq` $= 3.5\ GeV$, because according to (**??**), any track pair cut by this condition would anyway have been cut by the second occurence of the dcaTrackToPvx cut (because $p_{\perp_{V0}} \leq p_{\perp n} + p_{\perp p} \leq 3.5\ GeV$).

For security, a factor of 0.98 multiplies the $p_\perp$ limit in the first occurence of the cut. To sum up :

- First occurence : if $p_{\perp n} + p_{\perp p} \leq \sqrt{0.98} \times 3.5\ GeV$, apply the dcaTrackToPvx cut on both tracks
- Second occurence : if $p_{\perp_{V0}} \leq 3.5\ GeV$, apply the dcaTrackToPvx cut on both tracks

[0]Because of the opening angle of the V0 decay, the inequality is most often strict.

[0]Quod erat demonstrandum, not quantum electrodynamics ;-) .

4.4 XiFinder

4.4.1 Differences between Fortran and C++

The differences between the Fortran code and the C++ code are shown in Fig. **??**, on the half-detailed algorithm. The red lines show what has disappeared, either because of the new structure of the code, or because of the fact that we are using StEvent. The green lines show the parts that have been reshaped, for the same reasons as several parts have been removed.

Apart from these modifications, the code is a simple translation from Fortran to C++. This may change once we are convinced that the C++ code has no bug : we may then want to have the code more readable, or better organised, or we may even want to replace some calculation algorithms.

So far, the code is all in one block, for speed purposes, and the beginning and end of each former Fortran subroutine is indicated by commented lines. Once again, when we are sure that we don't need to compare the C++ and Fortran codes anymore, we'll probably remove all this.

Avoiding calls to subfunctions resulted in a duplication of a certain part of the code. Figure **??** show how the C++ code structure fits to the Fortran one, but let's detail the changes (green lines in Fig. **??**) one by one.

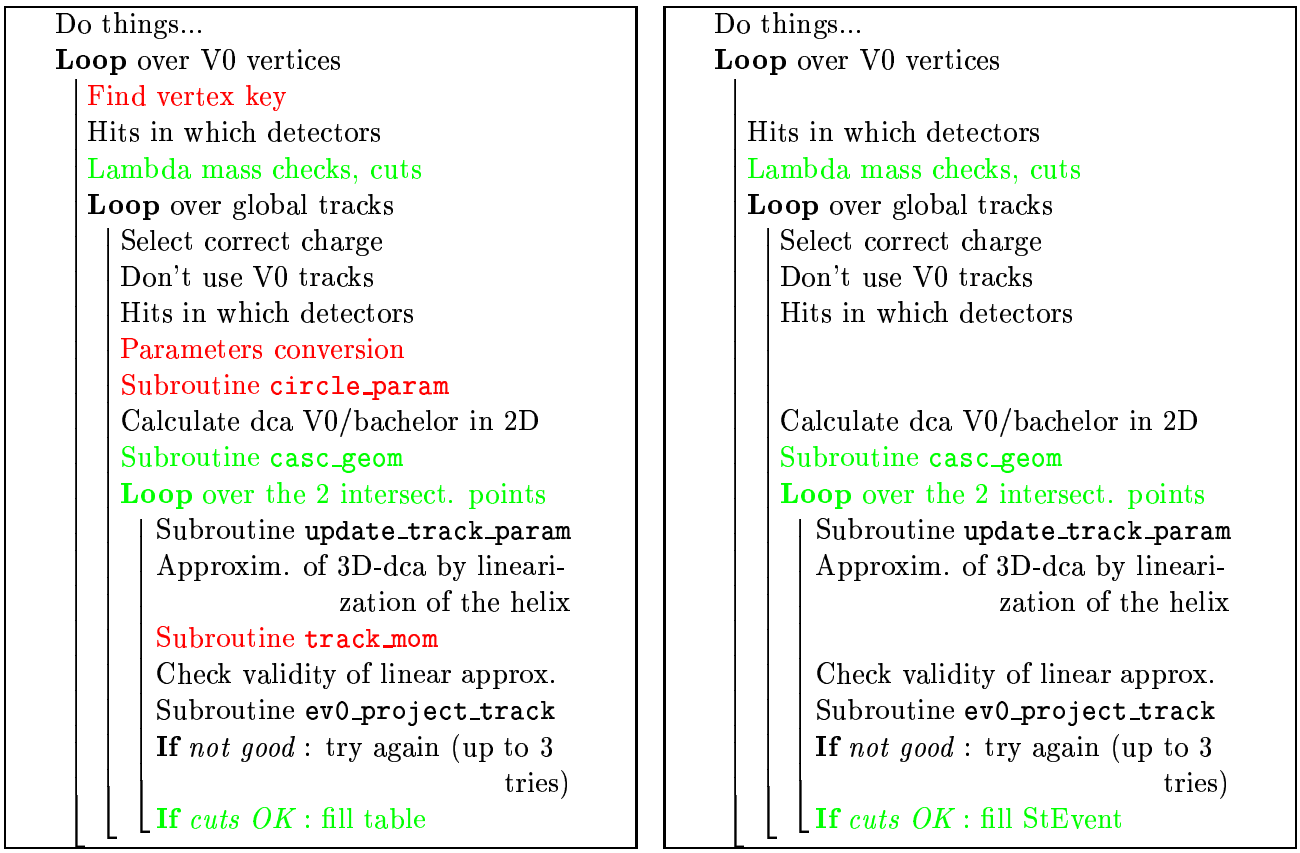


Fig. 5 : *Differences Fortran vs C++ in the XiFinder algorithm.*

The reshaping of the Lambda mass calculation consisted simply in calculating the invariant

mass using other parameters : taking the example of the Λ invariant mass, we have :

$$\begin{aligned} m_\Lambda^2 &= E_\Lambda^2 - \overrightarrow{p}_\Lambda^2 \\ &= (E_+ + E_-)^2 - (\overrightarrow{p}_+ + \overrightarrow{p}_-)^2 \\ &= E_+^2 + E_-^2 + 2E_+E_- - \overrightarrow{p}_+^2 - \overrightarrow{p}_-^2 - 2\overrightarrow{p}_+\overrightarrow{p}_- \end{aligned}$$

In Fortran, the E and $\overrightarrow{p}$ terms are grouped together, and the invariant mass is calculated with this formula :

$$m_\Lambda^2 = m_p^2 + m_\pi^2 + 2E_+E_- - 2\overrightarrow{p}_+\overrightarrow{p}_-$$

In C++, energies and momenta are kept separated, and the formula used is, as already written above :

$$\begin{aligned} m_\Lambda^2 &= (E_+ + E_-)^2 - (\overrightarrow{p}_+ + \overrightarrow{p}_-)^2 \\ &= \left(\sqrt{\overrightarrow{p}_+^2 + m_p^2} + \sqrt{\overrightarrow{p}_-^2 + m_\pi^2}\right)^2 - \overrightarrow{p}_\Lambda^2 \end{aligned}$$

To modify the structure of *casc_geom* and of the loop coloured in green in FIG. **??**, I've written the Fortran code as a set of for-loops, if-loops, goto's and blocks of instructions. The figures below show those reashapings : FIG. **??** shows the reshaping of *casc_geom*, and FIG. **??** shows the reshaping of the inside of the loop over the intersection points (between the bachelor's helix and the V0's straight line). In FIG. **??**, what is called *Block 5* is actually a huge part of the program, and it hasn't been reshaped.

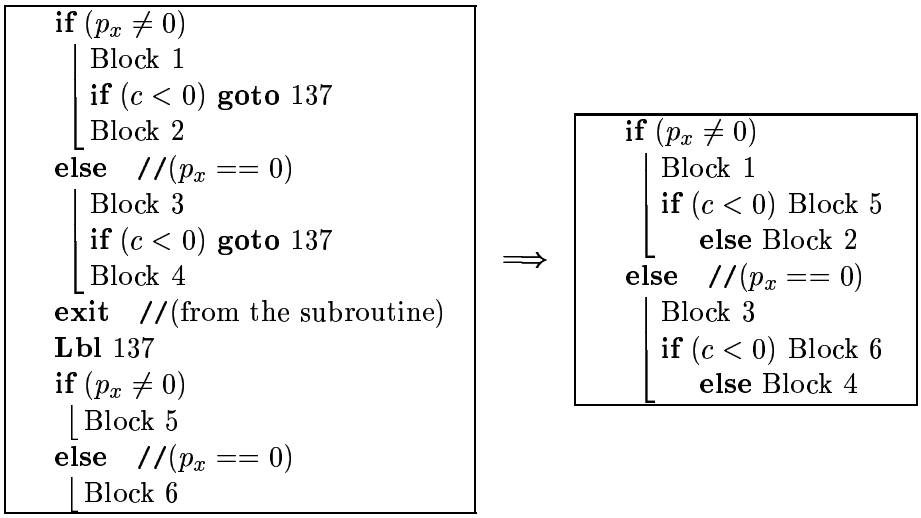


FIG. 6 : *Reshaping of* `casc_geom` *: Fortran to C++.*

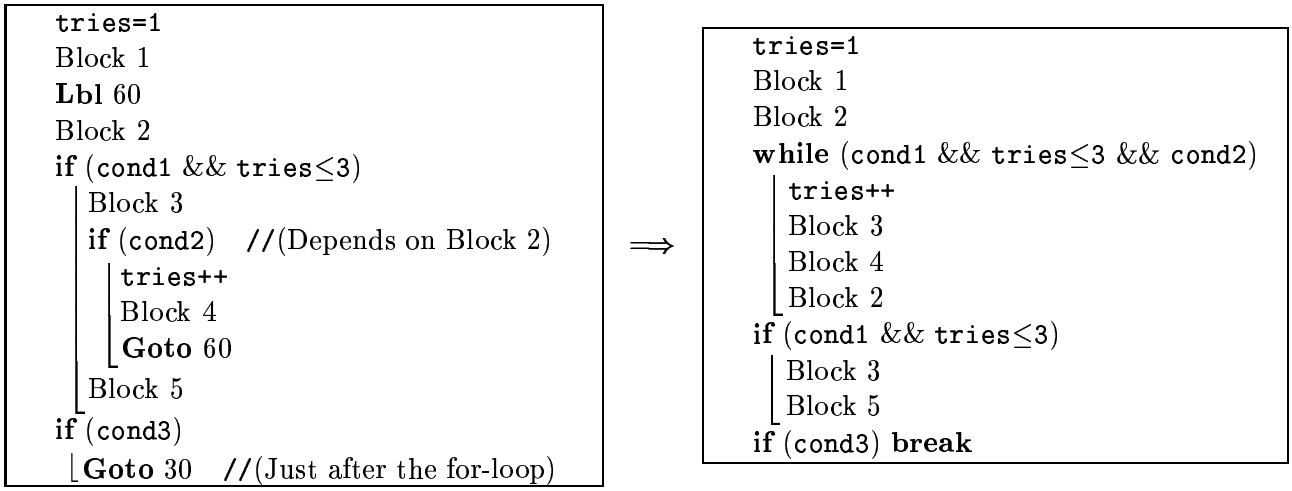


FIG. 7 : *Reshaping of the loop over the intersection points : Fortran to C++.*

4.4.2 Calculation of the intersection points

Now, here is how are calculated the coordinates of the intersection points *in the bending plane (xOy)* between the V0's straight line and the bachelor's helix (actually, of their projection in the bending plane). This is done in the code in the former subroutine `casc_geom`.

Let's call Δ the projection of the V0's trajectory, and $\mathcal{C}$ the circle that is the projection of the bachelor's trajectory. Their equations are :

$$\Delta : y = ax + b \qquad\qquad \mathcal{C} : (x - x_c)^2 + (y - y_c)^2 = R^2$$

Calling (x_0, y_0) a point on the V0's trajectory, and $\overrightarrow{p} = (p_x, p_y, p_z)$ its momentum, we obtain :

$$a = \frac{p_y}{p_x} \qquad\qquad b = y_0 - \frac{p_y}{p_x} x_0$$

Thus :

$$\Delta : y = \frac{p_y}{p_x}(x - x_0) + y_0$$

If we change the variables, with $X = x - x_c$ and $Y = y - y_c$, we have :

$$\Delta : Y = \frac{p_y}{p_x}(X + x_c - x_0) + y_0 - y_c \qquad\qquad \mathcal{C} : X^2 + Y^2 = R^2$$

Now, we can search the intersection points. If we call $\delta_x = x_c - x_0$ and $\delta_y = y_c - y_0$, we have to solve this system of equations :

$$\begin{cases} Y^2 &= R^2 - X^2 \\ Y^2 &= \left(\frac{p_y}{p_x}(X + \delta_x) - \delta_y\right)^2 \end{cases}$$

which, if we define $\alpha = p_y/p_x$ and $\beta = \alpha\delta_x - \delta_y$, and modify the equations, becomes :

$$\begin{cases} Y = \alpha(X + \delta_x) - \delta_y \\ X^2(\alpha^2 + 1) + 2\alpha\beta X + \beta^2 - R^2 = 0 \end{cases}$$

If the condition $R^2(\alpha^2 + 1) \geq \beta^2$ is true, then we have 2 solutions, that are :

$$\begin{cases} Y = \alpha(X + \delta_x) - \delta_y \\ X = \frac{\alpha\beta \pm \sqrt{R^2(\alpha^2+1) - \beta^2}}{\alpha^2+1} \end{cases}$$

The 2-D coordinates of these 2 points are stored in the code in variables called `xOut` and `yOut`.

4.4.3 Calculation of the dca between the V0 and the bachelor

The algorithm that calculates the dca between the V0 (a line) and the bachelor (a helix) is rather intuitive (see also FIG. **??**). This dca can't be calculated analytically, so the trick is to linearise the helix locally. This means that we will assume, for the dca calculation, that the helix is equal to its tangent at the intersection point between the helix and the V0 line.

Then, the position of the point of the tangent where the distance to the V0 line is the smallest is calculated. To check that the linearisation is not a too strong approximation, the disctance between

that point and the actual helix is calculated, and is required to be smaller than a certain fraction of the helix' radius. If this is true but if the distance is yet bigger than another fraction of the helix' radius (obviously smaller than the previous one), then the helix is linearised at the calculated point (instead of the intersection point in 2-D), and the calculation is re-done.

This is done 3 times, or less if the distance between the calculated point and the actual helix matches the second criterium after less than 3 loops. So all the candidates that match the first criterium are kept, and those not matching the second criterium are simply improved by trying 3 times to linearise the helix at a point that is closer to the actual point where the distance to the V0 line is the smallest.

The first part of the algorithm is illustrated by the figure **??**, which shows the projection in the plane (xOy) of the cascade geometry : that gives the circle $\mathcal{C}$ (projection of the helix) and the line Δ (projection of the 3-D line). M is one of the 2 intersection points (always in 2-D ; in 3-D, the helix and the line almost never intersect), C is the center of the circle, R its radius, and A is the projection of what is called the origin of the helix (it's most often the first hit point, or more exactly the point of the helix that is closest to the first hit, since the helix is a fit). Ψ is the angle between the x axis and the tangent to the circle in A. For lack of imagination, I'll keep the same names for the non-projected objects later ;-) (i.e. $\mathcal{C}$ for the helix, Δ for the V0 line and A for the origin (not its projection) of the helix).

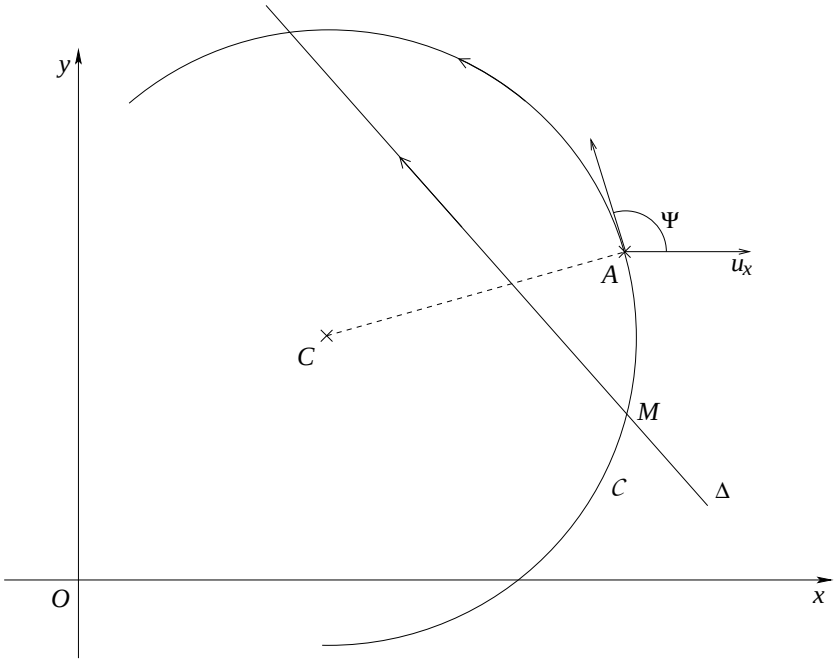


FIG. 8 : *Projection of the cascade geometry in the 2-D plane (xOy).*

The first part of the code that is run is the former subroutine `update_track_param`. Its role is simply to move the orgin of the helix from its former position A to its new position M (actually, the point of the helix that overlaps with M when projected to the (xOy) plane).

Here is a list of the various variables in this area of the code :

- `xi` : x coordinate of the origin
- `yi` : y coordinate of the origin
- `axb` : $\overrightarrow{CA} \cdot \overrightarrow{w}$, where $\overrightarrow{w}$ is the vector such as $\|\overrightarrow{w}\| = \|\overrightarrow{CM}\|$ and $(\widehat{\overrightarrow{CM}, \overrightarrow{w}}) = -\frac{\pi}{2}$

- `arg` : $\sin(\overrightarrow{CA}, \overrightarrow{CM})$
- `ds` : curvilinear length on the *circle* between A and M
- `dz` : $z_M - z_A$

And what follows is how to do the link between the code and the mathematical formulas :
In the code : `axb = (xi-xc)(yOut-yc) - (yi-yc)(xOut-xc)`

$$\overrightarrow{CA} = \left| \begin{array}{l} x_A - x_C \\ y_A - y_C \end{array} \right. \qquad \overrightarrow{CM} = \left| \begin{array}{l} x_M - x_C \\ y_M - y_C \end{array} \right. \qquad \overrightarrow{w} = \left| \begin{array}{l} y_M - y_C \\ -(x_M - x_C) \end{array} \right.$$

So $\overrightarrow{CA} \cdot \overrightarrow{w} = (x_A - x_C)(y_M - y_C) - (y_A - y_C)(x_M - x_C) =$ `axb`.
Now, let's try to find what is `axb` $= \overrightarrow{CA} \cdot \overrightarrow{w}$:
we know that $\|\overrightarrow{w}\| = \|\overrightarrow{CM}\|$, so :

$$\begin{aligned} \overrightarrow{CA} \cdot \overrightarrow{w} &= \|\overrightarrow{CA}\| \cdot \|\overrightarrow{w}\| \cos(\overrightarrow{CA}, \overrightarrow{w}) \\ &= \|\overrightarrow{CA}\| \cdot \|\overrightarrow{CM}\| \cos((\overrightarrow{CA}, \overrightarrow{CM}) + (\overrightarrow{CM}, \overrightarrow{w})) \\ &= R^2 \cos((\overrightarrow{CA}, \overrightarrow{CM}) - \tfrac{\pi}{2}) \\ &= R^2 \sin(\overrightarrow{CA}, \overrightarrow{CM}) \end{aligned}$$

Therefore, since `rsq` $= R^2$, we obtain `arg` $=$ `axb/rsq` $= \sin(\overrightarrow{CA}, \overrightarrow{CM})$.
`ds` is then defined as the angle (arcsin(`arg`)) multiplied by R, i.e. it is the curvilinear length on the circle from A to M.
And then, from the definition of the dip angle[0] , we obtain that `dz = ds.tan(dipAngle)` is $z_M - z_A$ (considering this time A and M as the points on the helix instead of on the circle).

At the end of the former subroutine `update_track_param`, a helix called `bachGeom2` is booked with the same parameters than the original bachelor helix taken from the track container, `bachGeom`, except for the origin that has been moved from A to M, and the angle Ψ that obviously changes when the origin moves (see FIG. **??**).

The next piece of code is the former subroutine `track_mom`, which just books the momentum of the bachelor taken in M in the variable `xOrig` (which contained the 3-D position of M a couple of code-lines before : since both usages don't overlap, the same StThreeVector can be used for both of them).

The next part of the code is the most difficult one to understand. It wasn't a subroutine in the Fortran code : that was part of the `exiam` function. Here is a list of the various variables in this area of the code :

- `pBach` : normalised momentum of the bachelor in M (so it's rather the (normalised) direction of the tangent to the helix in M)
- `dv0dotdb` : $\cos(\overrightarrow{\texttt{dpV0}}, \overrightarrow{\texttt{pBach}})$
- `diffc` : $\overrightarrow{MV}$, calling V the point where the V0 decays
- `denom` : $\cos^2(\overrightarrow{\texttt{dpV0}}, \overrightarrow{\texttt{pBach}}) - 1$
- `s2` : ehm... well... see the explanations below !
- `valid` : relative error due to the linearisation

So let's call V the position of the V0 vertex, i.e. the point where the V0 decays. As described p. **??**, we now linearise the helix, i.e. we now assume that the helix can be merged with its tangent in M. So an approximation of the point where the distance between the helix and the V0 line is the smallest is the point where the distance between the tangent to the helix and the V0 line is the smallest.

[0] For a detailed note about the helices' parameters, see the appendix A of the Star Class Library documentation.

Let $\mathcal{D}$ be the tangent to the helix in M, Δ being the V0 line, and let H_1 (resp. H_2) be the point on Δ (resp. on $\mathcal{D}$) where the distance to $\mathcal{D}$ (resp. to Δ) is the smallest.

Calling $\overrightarrow{u}$ the vector that drives Δ (i.e. $\overrightarrow{u} = \overrightarrow{\texttt{dpV0}}$) and $\overrightarrow{v}$ the vector that drives $\mathcal{D}$ (i.e. $\overrightarrow{v} = \overrightarrow{\texttt{pBach}}$), we can write H_1 and H_2 as :

$$H_1 = V + k_1 \overrightarrow{u} \qquad\qquad H_2 = M + k_2 \overrightarrow{v}$$

With the definition of H_1 and H_2 above, we can write that we search :

$$(H_1 \in \Delta, H_2 \in \mathcal{D}) \quad / \quad \overrightarrow{H_1H_2} \perp \Delta \quad \text{and} \quad \overrightarrow{H_1H_2} \perp \mathcal{D} \tag{2}$$

$$\Leftrightarrow \quad (H_1 \in \Delta, H_2 \in \mathcal{D}) \quad / \quad \overrightarrow{H_1H_2} \cdot \overrightarrow{u} = \overrightarrow{H_1H_2} \cdot \overrightarrow{v} = 0 \tag{3}$$

Given that

$$\begin{aligned} \overrightarrow{H_1H_2} &= M + k_2\overrightarrow{v} - V - k_1\overrightarrow{u} \\ &= \overrightarrow{VM} + k_2\overrightarrow{v} - k_1\overrightarrow{u} \\ &= -\overrightarrow{\texttt{diffc}} + k_2\overrightarrow{v} - k_1\overrightarrow{u} \end{aligned}$$

we can re-write the system (**??**) as

$$\begin{cases} \overrightarrow{H_1H_2} \cdot \overrightarrow{u} = -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{u} + k_2\overrightarrow{v} \cdot \overrightarrow{u} - k_1\overrightarrow{u} \cdot \overrightarrow{u} = 0 \\ \overrightarrow{H_1H_2} \cdot \overrightarrow{v} = -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{v} + k_2\overrightarrow{v} \cdot \overrightarrow{v} - k_1\overrightarrow{u} \cdot \overrightarrow{v} = 0 \end{cases}$$

which can also be written as

$$\begin{cases} -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{u} + k_2\cos(\overrightarrow{v}, \overrightarrow{u}) - k_1 = 0 \\ -\overrightarrow{\texttt{diffc}} \cdot \overrightarrow{v} + k_2 - k_1\cos(\overrightarrow{u}, \overrightarrow{v}) = 0 \end{cases}$$

because $\|\overrightarrow{u}\| = \|\overrightarrow{v}\| = 1$.
Solving this system, we obtain :

$$\begin{cases} k_1 = \frac{-\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{v}\cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{u})}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} \\ k_2 = \frac{\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{u}\cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{v})}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} \end{cases}$$

In the code, `s2` is calculated as :

```
s2 = (dpV0.X dv0dotdb - pBach.X) diffc.X + (dpV0.Y dv0dotdb - pBach.Y) diffc.Y + (dpV0.Z
dv0dotdb - pBach.Z) diffc.Z;
s2 = s2/denom;
```

which can be written as[0] :

$$\begin{aligned} \texttt{s2} &= \frac{(\overrightarrow{\texttt{dpV0}}\cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{\texttt{pBach}}) \cdot \overrightarrow{\texttt{diffc}}}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} \\ &= \frac{\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{u}\cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{v})}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} \\ &= k_2 \end{aligned}$$

So `s2` is the 3-D algebraic distance $\overline{MH_2}$ between M and H_2, the point of $\mathcal{D}$ that is closest to Δ.

[0] Given that $\texttt{dv0dotdb} = \overrightarrow{\texttt{dpV0}} \cdot \overrightarrow{\texttt{pBach}} = \cos(\overrightarrow{\texttt{dpV0}}, \overrightarrow{\texttt{pBach}})$. Thus $\texttt{denom} = \cos^2(\overrightarrow{\texttt{dpV0}}, \overrightarrow{\texttt{pBach}}) - 1 = \cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1$.

Then, `valid` is calculated as $\left|\texttt{s2}\sqrt{\texttt{pBach.X}^2+\texttt{pBach.Y}^2}\right|$, i.e. it's the distance in the (xOy) plane between M and H_2, as illustrated by figure **??**. The value `valid` itself is not very helpful to determine if the linearisation is a good approximation or not. The value that has to be looked at is actually the distance between H_2 and the circle in the 2-D plane, which is called d in figure **??**.

But d actually depends explicitely on `valid`, which means that an initial requirement on d can be transformed into a requirement on `valid`. This allows not to calculate d and saves some calculation time – at least it's the only reason I've found that would explain why the authors of the code have chosen to test `valid` rather than d ! The relation between d and `valid` is :

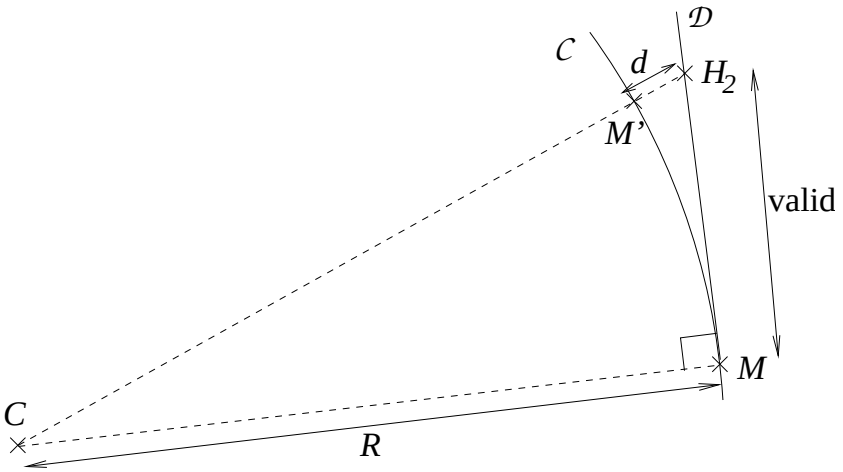
$$d = \sqrt{R^2 + \texttt{valid}^2} - R$$

$$\Leftrightarrow \quad \frac{d}{R} = \sqrt{1 + \left(\frac{\texttt{valid}}{R}\right)^2} - 1 \tag{4}$$

There are 2 conditions on $\frac{\texttt{valid}}{R}$. Let's call these two values $valid_1$ and $valid_2$, with $valid_1 < valid_2$. The original algorithm (I may change that, and will try to update this documentation if so) throws away any Xi candidate for which $\texttt{valid} > valid_2$, and keeps all the other ones. But if $\texttt{valid} \in [valid_1; valid_2]$, then another part of the algorithm, which I describe in the next paragraph, is run. It consists in improving the quality of the linearisation by linearising the helix at another point. This improvement is tried at most 3 times. Basically, this means that a linearisation that gives $\texttt{valid} > valid_2$ means that it's hopelessly bad ; when $\texttt{valid} < valid_1$ it means that the linearisation is good enough and doesn't need to be improved ; and when $\texttt{valid} \in [valid_1; valid_2]$, the linearisation is improved but it actually doesn't matter if the criterium $\texttt{valid} < valid_1$ is not reached : the candidate is kept anyway. As far as I've seen during the tests, only a very few proportion of the candidates need 3 passes in the loop[0], so requiring more than 3 passes is indeed not necessary.

This table shows the numerical values of $valid_1$ and $valid_2$ used in the code (first line) and their equivalent for the more interesting variable $\frac{d}{R}$, calculated with (**??**).

$0.001\ R <$	`valid`	$< 0.02\ R$
$5.10^{-7}\ R <$	d	$< 2.10^{-4}\ R$
$0.0005\ \% <$	$\frac{d}{R}$	$< 0.02\ \%$



[0]Result obtained over 1 `Au-Au` 200 *GeV* central event : over 73 269 bachelor, neglecting those which have only 1 intersection point, 45 500 have 2 intersection points, i.e. 62 % of them (and therefore 38 % have no intersection points). Among the 118 194 dca calculations of those 62 %, 77.0 % of them don't need a better linearisation, 21.7 % need to go once in the loop, 0.8 % need to go twice in the loop, and 0.5 % go 3 times in the loop (this latter percentage, unike the former ones, is the number of candidates that need only 3 passes added to the number of candidates that would need more).

FIG. 9 : *2-D plane evaluation of the quality of the approximation made by linearising the helix.*

So when `valid` $\in [valid_1; valid_2]$, here is the piece of code that is run :

- Former subroutine `ev0_project_track` : calculates the coordinates (in the (xOy) plane) of the point M' defined as the intersection of the circle $\mathcal{C}$ and the line (CH_2) (see FIG. **??**)
- Former subroutine `update_track_param` : moves the origin of the helix from M to M' (as previously done from A to M)
- Former subroutine `track_mom` : calculates the momentum of the bachelor in M' (as previously done in M)
- Block that calculates `s2` and `valid` : calculates a new `s2` and `valid`, whose value will be checked to see if one more pass in this loop is necessary

The 3 last blocks are exactly the same as those already described above, so I'll only describe the former subroutine `ev0_project_track` : the list below is made of the variables that are used in this area of the code :

- `batv` : 3-D coordinates of H_2
- `dtmp` : $x_C - x_{H_2}$
- `atmp` : $y_C - y_{H_2}$
- `ctmp` : slope of the line (CH_2)
- `yy` : $y_{M'} - y_C$
- `zz` : $x_{M'} - x_C$
- `xAns` : temporary variable that I'll remove soon...
- `yAns` : temporary variable that I'll remove soon...
- `xOut` : $x_{M'}$ (contained x_M before)
- `yOut` : $y_{M'}$ (contained y_M before)

The calculation of $(x_{M'}, y_{M'})$ is simple : since $\texttt{ctmp} = \frac{x_C - x_{H_2}}{y_C - y_{H_2}}$ is the inverse of the slope of (CH_2) the equation of (CH_2) is :

$$(CH_2) : y = \frac{1}{\texttt{ctmp}}(x - x_C) + y_C$$

and therefore, setting $x' = x_{M'} - x_C$ and $y' = y_{M'} - y_C$, M' is such as :

$$\begin{cases} y' = \left(\frac{1}{\texttt{ctmp}}\right) x' \\ x'^2 + y'^2 = R^2 \end{cases}$$

Solving this system gives :

$$\begin{cases} y' = \pm \frac{R}{\sqrt{1+\texttt{ctmp}^2}} \\ x' = \texttt{ctmp} \cdot y' \end{cases}$$

x' and y' are respectively `zz` and `yy` in the code, and the "signus dilemma" is solved by the if-loop on the sign of `atmp`.

At the end of the while-loop, the candidates for which `valid` $> valid_2$ are simply thrown away, and all the other ones are kept, even if `valid` $> valid_1$. The value of `s2` calculated during the last loop is kept, and `s1` (which is the k_1 of equation (**??**) (p. **??**) and of the following ones) is calculated as

$$\texttt{s1} = \frac{-\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{\texttt{pBach}} \times \texttt{dv0dotdb} - \overrightarrow{\texttt{dpV0}})}{\texttt{dv0dotdb}^2 - 1} = \frac{-\overrightarrow{\texttt{diffc}} \cdot (\overrightarrow{v}\cos(\overrightarrow{u}, \overrightarrow{v}) - \overrightarrow{u})}{\cos^2(\overrightarrow{u}, \overrightarrow{v}) - 1} = k_1$$

Then, the 3-D coordinates of H_1 are calculated and stored in `v0atv`, just like the coordinates of H_2 are stored in `batv`. Once this is done, we check that $\overrightarrow{H_1V}$ goes roughly in the same direction as $\overrightarrow{p_{V0}}$, i.e. that the V0 points away from the Xi vertex newly found. This is done via the variable `check`, that is exactly $\overrightarrow{H_1V} \cdot \overrightarrow{p_{V0}}$.

If `check` is positive, it means that the V0 points to the opposite direction than where the Xi vertex is, and the rest of the algorithm – run only in that case – can be roughly summed up as something that looks like that :

```
dca = ||v0atv − batv||                    // dca = H1H2
xpp = (v0atv−batv)/2                      // xpp = 3-D coordinates of the middle of [H1H2]
rv = ||xpp − xPvx||                       // rv = Xi decay length
if (cuts are OK)
    pXi = pV0 + xOrig                     // p_Xi = p_V0 + p_Bach,H2
    check = (xpp - xPvx) . pXi            // Check that PvxXivtx · p_Xi ≥ 0, i.e. that the
    if (check ≥ 0)                        //        reconstructed Xi points away from the Pvx
        iflag=0 if the dcaXiPvx (cut) is OK
    if (iflag=0)
        Calculates the kinematic variables
        Store the candidate
```

4.4.4 Detailed algorithm and cuts

The detailed algorithm is actually explained in the section **??** concerning the calculation of the dca, p. **??**. Yet, in the latter paragraph, no overview of the algorithm is given and the cuts are not listed. This is the purpose of this short paragraph, and is summed up in FIG. **??**.

Blablabla.

```
Loop over V0 vertices
    Apply cuts on the V0 and V0 daughters
    Loop over global tracks
        If track has wrong charge : next
        If track already used in the V0 : next
        Find DCA between V0's straight line and track's helix
        Apply cuts
        If good candidate : store
```

FIG. 10 : *Cuts and algorithm of the XiFinder.*

5 Detectors

Blablabla

6 Rotating and like-sign

These options have been implemented for analysis purpose, and have been thought to be a plug-and-play code, avoiding private dirty versions ;-) and waste of time for those who wish to use such methods and would have had to code them themselves.

6.1 Like-sign analysis

For a decay channel $A \longrightarrow B + C$, the like-sign method consists in reconstructing A by associating B and $\overline{C}$ rather than B and C. Its name comes from the usage in decays such as $\Lambda \longrightarrow p^{(+)} + \pi^-$ (using like-sign, p would be combined with π^+, a particle that has the same sign), but let's extend this usage to the Ξ-like decays, e.g. $\Xi^- \longrightarrow \Lambda^0 + \pi^-$. Since the Λ is neutral, associating a Λ with a π^+ instead of a π^- isn't doing like-sign strictly speaking, but I will call this like that.

Like-sign analysis has been implemented only in the XiFinder, not in the V0Finder. The reason why is that one of the Xi decay is neutral, and combining a stright line with a positive helix rather than a negative one doesn't change anything. But things are different for a V0 decay : in such a case, "like-singing" means combining 2 helices of same charge, instead of opposite charge, and, although I've never checked that or looked at distributions made by somebody else, I'd bet that things like the combinatoric, the dca distribution, etc... are changed, which means that the background built with this method would be different than the real background, because the same cuts are applied to distributions that don't have the same shape. So in the case of the V0 decays, the rotating method is probably safer than like-sign.

As said previously, the like-sign method in the XiFinder consists in finding candidates build with one of the daughters being the antiparticle of the expected daughter. So we find and store $\Lambda + \pi^-$ (resp. K^- for the Ω) and $\overline{\Lambda} + \pi^+$ (resp. K^+).

In the post-reconstruction analysis codes, one should be very careful when using like-signed candidates, because the charge of a particle is determined with the charge of the bachelor[0]. So what your code will assume is an Ω^- is a particle made of a $\pi-$ and a $\overline{\Lambda}$ instead of a Λ. So when applying the cuts, all the functions that make a hypothesis on the particle identification should be changed (e.g. `massLambda()` $\rightarrow$ `massAntiLambda()`).

Deciding whether like-sign should or shouldn't be used is done by using the function `StV0FinderMaker::SetLikesignUsage`. The 2 possible values are :

- `kLikesignUseStandard` $= 0$: the standard reconstruction with no like-sign is performed,
- `kLikesignUseLikesign` $= 2$: the like-sign reconstruction is done.

Section ?? p.?? explains how to use these functions.

In the code, like-sign analysis is done in a very easy way :

- When checking that-and-whether the V0 is a Λ or a $\overline{\Lambda}$, -1 is stored in variable `charge` if it's a Λ (i.e. a Ξ^- or Ω^- candidate will be built), +1 if it's a $\overline{\Lambda}$ (i.e. a $\overline{\Xi}^+$ or $\overline{\Omega}^+$ candidate will be built).
- Then, the variable `charge` is transformed according to this formula :
  `charge=-(useLikesign-1)*charge;` :
  if no like-sign has been asked, the variable is unchanged, whereas if like-sign has been required, the sign of `charge` is changed.
- The XiFinder eventually loops over tracks whose charge is such as `charge` $\times$ `track.charge` > 0.

And that's all !

[0]This can be found in `StRoot/StStrangeMuDstMaker/StXiMuDst.cc`, in function `StXiMuDst::FillXi(StXiVertex* xiVertex)`.

6.2 Rotating analysis

For a decay channel $A \longrightarrow B + C$, the rotating method consists in reconstructing A by associating B with C' rather than C, where C' is the track of a C-like particle whose parameters have been changed. The various possible changes are :

- Rotating : a track is rotated by 180° around the axis that is parallel to (Oz) and that goes through the primary vertex,
- Symmetry : a track is transformed into its symmetric with respect to the (xOy) plane,
- Rotating + symmetry : doing both transofmations together is equivalent to taking the symmetric of the track with respect to the primary vertex.

Rotating – which will refer from now to all 3 methods described in the previous paragraph – hasn't been implemented in the V0Finder yet, but will be some day (I haven't received any request yet ;-)).

Deciding whether one of these methods should be used and which one is done by using the function `StV0FinderMaker::SetRotatingUsage`. The usage is explained in section **??** p. **??**.There are 4 possible values, which are :

- `kRotatingUseStandard` $= 0$: the standard reconstruction with no rotating is performed,
- `kRotatingUseRotating` $= 1$: the bachelor tracks are rotated,
- `kRotatingUseSymmetry` $= 2$: the bachelor tracks are "symmetrised" with respect to the bending plane,
- `kRotatingUseRotatingAndSymmetry` $= 3$: the bachelor tracks are "symmetrised" with respect to the primary vertex.

Unlike with like-sign, nothing has to be changed in the post-reconstruction analysis codes.

Here is a mathematical description of how the various rotating-like methods are performed (see footnote **??** p. **??** about the helices' parameters) : the table below shows how the various helix parameters are modified depending on the method used.

Original helix		Rotating	Symmetry	Both
Charge	c	c	c	c
Angle	Ψ	$\Psi + \pi$	Ψ	$\Psi + \pi$
Curvature	κ	κ	κ	κ
Dip angle	λ	λ	$-\lambda$	$-\lambda$
X origin	x_0	$2x_{Pvx} - x_0$	x_0	$2x_{Pvx} - x_0$
Y origin	y_0	$2y_{Pvx} - y_0$	y_0	$2y_{Pvx} - y_0$
Z origin	z_0	z_0	$2z_{Pvx} - z_0$	$2z_{Pvx} - z_0$
Helicity	h	h	h	h
X momentum	p_x	$-p_x$	p_x	$-p_x$
Y momentum	p_y	$-p_y$	p_y	$-p_y$
Z momentum	p_z	p_z	$-p_z$	$-p_z$

In the code, all rotating methods calculations are achieved in one shot, thanks to the pre-definition of a couple of interesting variables which I describe below. In the code, once the bachelor helix is moved, nothing else is changed by the use of a rotating-like method. So what is done is simply the booking (and then usage) of a StHelixModel called `bachGeom` from both the initial parameters of the helix and the "interesting variables", whose value is set at the beginning of the XiFinder, before the loop on the bachelor tracks.

Here is how the helix parameters are modified before their storage in `bachGeom` :

charge $\longrightarrow$ charge
helicity $\longrightarrow$ helicity
curvature $\longrightarrow$ curvature
psi $\longrightarrow$ psi + **cstPsi**
dipAngle $\longrightarrow$ **epsDipAngle** $\times$ dipAngle
origin.X $\longrightarrow$ **cstOrigin.X** + **epsOrigin.X** $\times$ origin.X
origin.Y $\longrightarrow$ **cstOrigin.Y** + **epsOrigin.Y** $\times$ origin.Y
origin.Z $\longrightarrow$ **cstOrigin.Z** + **epsOrigin.Z** $\times$ origin.Z
momentum.X $\longrightarrow$ **epsMomentum.X** $\times$ momentum.X
momentum.Y $\longrightarrow$ **epsMomentum.Y** $\times$ momentum.Y
momentum.Z $\longrightarrow$ **epsMomentum.Z** $\times$ momentum.Z

The "interesting variables" are written in bold, and the values they are given are listed in the table below (an empty space means that the value is the same as for "no rotating") :

Variable	No rotating	Rotating	Symmetry	Both
`cstPsi`	0	π		π
`epsDipAngle`	+1		-1	-1
`cstOrigin.X`	0	$2x_{Pvx}$		$2x_{Pvx}$
`cstOrigin.Y`	0	$2y_{Pvx}$		$2y_{Pvx}$
`cstOrigin.Z`	0		$2z_{Pvx}$	$2z_{Pvx}$
`epsOrigin.X`	+1	-1		-1
`epsOrigin.Y`	+1	-1		-1
`epsOrigin.Z`	+1		-1	-1
`epsMomentum.X`	+1	-1		-1
`epsMomentum.Y`	+1	-1		-1
`epsMomentum.Z`	+1		-1	-1

The combination of the values in this table and the transformations listed just above it give the mathematical transformations listed in the table p. **??**. This avoids a check of the rotating choice by an if-loop inside the for-loop on the tracks, and thus time-consuming jumps in the code.

7 Cuts values

Blablabla

8 How to use the V0/XiFinder

Blablabla

9 Tests

Blabla.